University of Saskatchewan
Department of Computational Science

Cmpt 340
Final Examination

December 18, 1995

Time: 3 hours
Total Marks: 114

Professor: A. J. Kusalik
Closed Book[†]

Name: ______________________________

Student Number: ____________________

Directions:

Answer each of the following questions in the space provided in this exam booklet only. If you must continue an answer (e.g. in the extra space on the last page, or on the back side of a page), make sure you clearly indicate that you have done so and where to find the continuation.

Ensure that all written answers are legible; no marks will be given for answers which cannot be deciphered. Where a discourse or discussion is called for, please be concise and precise. If you find it necessary to make any assumptions to answer a question, please state the assumption with your answer.

Marks for each major question are given at the beginning of that question.

The examination begins on the next page. Good luck.

A.	_____/7	F.	_____/4	K.	_____/6
B.	_____/26	G.	_____/4	L.	_____/8
C.	_____/3	H.	_____/6	M.	_____/9
D.	_____/6	I.	_____/8	N.	_____/14
E.	_____/3	J.	_____/10		
				Total:	______/114

[†] Closed book, except for one optional 8.5x11 inch quick reference sheet ("cheat sheet") of the student's own compilation.

A. ***(7 marks)***

Answer each of the following questions with a very short and precise response.

1. What is the best programming language?

2. Would you expect a common programmable calculator to use interpretation or compilation to perform its (program-based) operations?

3. The ability to write meta-interpreters in Prolog (an hence modify the language's semantics or define new, related languages) is an example of satisfying what programming language criterion?

4. Consider the parameter-passing mechanism of C or C++. Is the binding of actual to formal parameters a *static* or a *dynamic* binding?

5. Consider the following loop (in a Algol-like language):

```
i := n; sum := 0;
while i do
    sum := sum+i;
    i := i-1
od
```

Which of the following is (are) the loop invariant(s) for the loop?

a) `sum = i + . . . + n`

b) `sum = (i+1) + . . . + n and i > 0`

c) `sum ≥ 0 and i ≥ 0`

6. The difference in meaning of the C++ statements

```
const char *s = "hello world";
```

and

```
char const *s = "hello world";
```

is an example of a violation of what programming language principle?

7. Prograph incorporates and exemplifies several programming language paradigms. Two of them are paradigms not well-illustrated by, or not present in, the other languages we have studied. One of these is the visual programming paradigm. What is the other one?

B. ***(26 marks)***

Each of the following questions is comprised of a number of statements regarding a particular subject area covered in class. For each set of statements, identify (and indicate) the statement which is false in that set. Each set of statements is headed by the subject area to which they relate.

1. History of programming languages

(a) A vast number of special-purpose languages were designed in the 1960s and 1970s, but often these were only used by the small group who implemented them.

(b) Languages such as FORTRAN, COBOL, and LISP, although developed in the 1950s and early 1960s, are still in widespread use.

(c) PL/I attempted to provide a general multi-purpose language by combining many of the features of FORTRAN, ALGOL 60, and IBM Assembly Language. However, it proved to be too large and unwieldy.

(d) Ada, Modula-2, and C++ are among the most important of the modern generation of general-purpose imperative languages.

(e) High-level languages developed from the need to make programming easier for the non-specialist programmer and to make languages independent of specific computer hardware.

2. History of programming languages

(a) BASIC became popular due to its simplicity and the fact that its interpreter would (typically) fit into the memory of small machines.

(b) FORTRAN introduced many new concepts into programming languages including recursion, block structure, syntax definition, and structured elements.

(c) PASCAL became a major teaching language, and has had a beneficial influence on programming style.

(d) The language C++ attempts to combine the advantages of object-oriented and conventional imperative languages.

3. Areas of application for programming languages

(a) Ada has proved to be the most successful systems programming language.

(b) COBOL was the first effective high-level language for commercial data processing.

(c) LISP has been the dominant language in artificial intelligence for three decades.

(d) SNOBOL4 was (is) a languages specifically designed for string manipulation

(e) Prolog is well-suited to applications involving database queries.

(f) The features of APL make it a good language for scientific and engineering applications.

4. Programming language paradigms

(a) There are at present four major programming (language) paradigms: imperative, object-oriented, functional, and logic. Emerging paradigms are concurrent (parallel) and visual languages.

(b) LISP is oriented towards the manipulation of symbols, rather than numbers, and is close to being a functional language.

(c) Object-oriented programming has received much attention in recent years, and the main object-oriented languages are Smalltalk, Eiffel, and C++.

(d) A logic programming language is any language based solely on Horn-clause logic and (Robinson's) resolution. Prolog is the most common logic programming language.

(e) In dataflow programming languages, data can be considered active. Data flow through a program and activate each instruction as soon as all the required input data have arrive at/for that instruction.

5. Programming language criteria

(a) A programming language should be simple, have high expressive power and language constructs that are consistent — that is, they should have a similar effect wherever they appear.

(b) It is important to have a proper balance between the introduction of powerful language features and their ease of implementation.

(c) A language should have a single official definition to which all implementors adhere.

(d) Languages should support the detection of as many errors as possible at compile time.

(e) "Friend functions" are a feature of programming languages that support information hiding.

(f) Since any serious program is read many more times during its lifetime than it is written, readability is more important than writability.

(g) Two features in a language are orthogonal if they can be combined freely without restricting each other in any way.

6. Implementation techniques and concepts

(a) An interpreter is a program that can read, analyze, and execute constructs in a language. A compiler, on the other hand, translates these constructs into machine instructions, rather than executing them.

(b) Final program composition is generally done using the (system's) linker.

(c) Attributes that a variable could have include its name, type, location (reference), and current value.

(d) A late binding time for an object's attributes leads to efficient programs, while earlier binding leads to more flexibility.

(e) If the amount of storage required by all local objects can be determined at compile time, it is possible to implement a language so that the relative position of each object within a data area is known at compile time.

7. Semantics

(a) There are two approaches to formal semantics: denotational and axiomatic. There are also two approaches to informal semantics: natural language and operational.

(b) In denotational semantics, language constructs are modelled by appropriate mathematical objects whose meaning is already known.

(c) In axiomatic semantics, rules of inference are used to deduce the effect of executing a language construct.

(d) In many language reference manuals, the semantics of language constructs are not described formally; rather, they are given in a natural language such as English.

(e) Operational semantics is a method of describing the meaning of language constructs in terms of their effect on a machine, often an ideal or virtual machine.

8. Logic-programming languages

(a) The logic programming model separates the statement of a problem from the method used to solve it.

(b) A major step in a Prolog computation is the matching of a goal with the head of a clause. This matching is performed by a restricted form of unification.

(c) "CLP" does not refer to a single language, but rather to a family of languages.

(d) Applications of CLP include operations research, scheduling, and modelling.

(e) At each step in the solution of a goal in CLP, the accumulated set of constraints must be checked for satisfiability.

(f) In a constraint logic programming language, there exists a "domain of interpretation" from which (program) variables can be drawn. The properties of a particular CLP language depend on the particular "domain of interpretation" chosen.

9. Object-oriented languages

(a) The object-oriented terminology is that objects send messages to one another. On receipt of a message at run time, an object decides which method it will use in response.

(b) There can be a conflict between the aims of inheritance and the programming language criterion of information hiding.

(c) The object-oriented paradigm not only deals with programming, it is also a paradigm for designing systems; in particular, large and complex ones.

(d) The most important concepts in object-oriented languages are objects, classes, inheritance, polymorphism, and dynamic binding.

(e) In dynamic binding, the decision about which operation (method, member function) to use must be made before runtime.

10. C++

(a) The meaning of an operator can depend on its operands. Such an operator is said to be overloaded.

(b) In C++ data abstraction is provided by the *struct* construct, which provides a facility closely modelled on the concept of classes in SIMULA67 and Smalltalk.

(c) A derived class may access protected information in the base class which is not visible to clients of the base class.

(d) C++ encourages reuse of classes by allowing separate compilation of separate files and separate specification of declaration and implementation files.

11. Modern functional languages (e.g. Gofer)

(a) In functional languages, functions themselves are treated as first-class objects.

(b) Functions which can take other functions as parameters or which return functions as results are called higher-order functions.

(c) As side effects cannot occur in a purely functional language, calling a function with a certain parameter will always give the same answer. This is called referential transparency.

(d) Gofer is strongly-typed. The programmer, however, does not have to specify all the types (of Gofer functions), as the language has a type inference system that can (usually) determine what the types must be.

(e) Functions are defined in Gofer by a set of alternative definitions. The appropriate definition is chosen by means of unification.

(g) Currying is an important concept in functional languages.

12. Modern functional languages

(a) Lazy evaluation corresponds to call-by-need, and produces termination whenever termination is possible.

(b) Normal-order reduction first evaluates the arguments of a function, and then applies the function. Applicative-order reduction applies the function to the unevaluated arguments, and only evaluates the arguments when needed.

(c) Eager evaluation corresponds to call-by-value, and may cause redundant evaluation or unnecessary nontermination.

(d) Normal-order reduction is the basis of lazy evaluation.

(e) Lazy evaluation makes it possible to deal with infinite data structures.

13. Prograph

(a) Methods (for a class) are specified in a high-level pictorial language. Each icon has syntactic meaning.

(b) The method editor in Prograph is context aware (or context sensitive). As a result, a *command-click* will generate different constructs (icons) depending on where the mouse is located when the *command-click* is given.

(c) There are four different ways of referencing a method in Prograph: explicit-class, data-determined, universal, and context-determined.

(d) Because iteration is not easily described in a visual paradigm, repetition is only achieved by recursion in Prograph (just as it is in Gofer and Prolog).

C. ***(3 marks)***

Indicate which of the following classes of languages would be considered ***declarative*** languages.

___ dataflow languages

___ logic-programming languages

___ meta-interpreted languages

___ object-oriented languages

___ functional languages

___ interpreted languages

___ virtual-machine languages

D. ***(6 marks)***

Answer each of the following questions with a short, precise answer. The questions are based on specific examples used in class materials.

1. Recall the following specification of denotational semantics for the *while-do* loop statement (in a language like Pascal):

```
$M_l$( while B do L, s ) ≡
    if $M_b$(B,s) = **undef**
        then **error**
        else if $M_b$(B,s) = false
            then s
            else if $M_{sl}$(L,s) = **error**
                then **error**
                else $M_l$( while B do L, $M_{sl}$(L,s) )
```

a) What is the semantic domain (i.e. the range) of the semantic function M_l?

b) Why are the symbols **undef** and **error** specially designated, i.e. shown in boldface?

2. Recall the formulation of the factorial function in the language CLP(*Q*) (actually, the CLP(*Q*) extension to SICStus Prolog). That program could be rewritten in CLP(*R*) as follows:

```
/* factorial function in CLP(R) */
fact( 0, 1 ).
fact( 1, 1 ).
fact( N, F*N ) :- N > 1, fact( N-1, F ).
```

One of the important characteristics of logic programming languages, CLP languages included, is that queries can be given with varying patterns of instantiated and uninstantiated arguments. What would the output be if the above program was queried with

```
?- fact( X, Y ).
```

I.e. what output would be generated if the factorial program (in CLP(*R*)) was given a query with all arguments uninstantiated? (Try to be as complete as possible in your answer).

E. ***(3 marks)***
With any formal description of the semantics of a language, the description must be written in some language. We could call this the *defining* language to distinguish it from the defined language. For each of the methods of specifying semantics surveyed in class, what is the defining language?

1. Operational Semantics

2. Axiomatic Semantics

3. Denotational Semantics

F. ***(4 marks)***
Write lambda expressions (λ-expressions) for the following functions typically found on calculators. Also show an example application of each λ-expression; i.e. give an example showing application of your λ-expression to (an) argument(s).

1. x^2

2. Fahrenheit to Celsius conversion
(Recall that to do this conversion, one subtracts 32 and then multiplies by 5/9).

G. ***(4 marks)***
Overloaded operators are not unique to C++. They occur in other languages as well, including C and Pascal. Give two operators in C or Pascal which are overloaded, and describe why they can be considered to be overloaded.

H. ***(6 marks)***

1. Languages such as Gofer and Prolog do not have pointers (i.e. data types which are pointer-to-*type*). Why is this the case?

2. Why do you think that lists are the primary data structure in most functional languages? (In contrast, arrays are the primary data structures in imperative languages.)

I. ***(8 marks)***

Answer the following two questions relating to program language criteria with concise and precise answers.

1. C++ has two different mechanisms for deallocating objects that are no longer used. (This is different than languages like Eiffel, for example, which has only one). C++ has automatic deallocation of declared objects, but dynamically-created objects must be deallocated by the programmer. From the standpoint of programming language criteria, do you think this is good or bad? Why? For example, what programming language criteria would be satisfied or violated? Explain how.

2. Algol 60 had no records and no pointers. A programmer who wanted to implement a linked list of real numbers declared two arrays, **`real`** `value[100]` and **`integer`** `next[100]`, as well as an integer **`integer`** `nrecs`. The nth record corresponds to the pair of values `value[n]` and `next[n]`, where `value[n]` holds the value and `next[n]` holds the index of the next record in the linked list. The integer `nrecs` counts how many records exist.

   Criticize the above construction based on the programming language criteria given in the course. Assume that you are trying to present a contrary argument to a person who thinks that the above construction is "just fine", and that records and pointers are semantic sugar "real programmers" don't need.

J. ***(10 marks)***
Most programming languages have the construct of "variables". However, the characteristics of variables vary greatly among languages. In fact, one distinguishing feature of programming languages is whether they have variables, and if they do, the characteristics of those variables.

1. One of the interesting aspects of constraint logic programming languages (CLPs) is the characteristics associated with "variables" in these languages. CLPs have a construct of "variables" that has similarities with "variables" in other languages, as well as differences. To this end, briefly contrast (describe the differences between and similarities of) the construct of a variable in each of following pairs of languages:

   a) CLP and C++

b) CLP and Prolog

c) CLP and Gofer

2. Are there variables in Prograph in the sense of variables in imperative languages (e.g. C, C++, or Pascal)? Why or why not?

K. ***(6 marks)***

Both Prolog and Gofer have list structures; in both we can talk of the "head" and "tail" of a list. The two languages even have much the same syntax for lists. For example, in both `[1,2,3]` is a list consisting of the three numbers 1,2, and 3. Also, the Gofer syntax `(a:x)` is equivalent to Prolog's `a.x` or `[a|x]`.

It is easy to define functions in Gofer to manipulate lists. For example,

```
head (a:x) = a
```

and

```
tail (a:x) = x
```

can be used to obtain the head and tail of a list, respectively. In Prolog, analogous predicates can be defined just as simply:

```
% head( List, Head ) is true if Head is the head of list List
head( [Head|_Tail], Head ).

% tail( List, Tail ) is true if Tail is the tail of list List
tail( [_Head|Tail], Tail ).
```

Consider the Gofer functional expressions

```
head [1,2,3]
tail [1,2,3]
```

and the Prolog queries

```
?- head( [1,2,3], Head ).
?- tail( [1,2,3], Tail ).
```

What are the conceptual differences (Gofer versus Prolog) in the computations invoked by the above statements (the Gofer expressions and the Prolog queries)?

L. ***(8 marks)***

1. Write a function *rotate* in Gofer that, given an integer *n* and a list *l*, rotates the elements of *l* left by *n* places. For example

   rotate 0 [a,b,c] = [a,b,c]

   and

   rotate 2 [a,b,c,d,e] = [c,d,e,a,b]

   (You can assume that *n* is a nonnegative integer).

2. What is the type of the *rotate* function above?

3. Write a predicate *rotate(n,list,rotlist)* in Prolog which is true if list *rotlist* is the list *list* rotated left by *n* places. For example

   rotate(0, [a,b,c], [a,b,c])

   and

   rotate(2, [a,b,c,d,e], [c,d,e,a,b])

   are both true. (Again you can assume that *n* is a nonnegative integer).

M. (9 marks)

1. One of the ways that we categorized programming languages in Cmpt340 was according to application area; i.e. for a specific application areas, certain languages were better suited and more popular. What type of application area do you think that the language Prograph is best-suited for? I.e. if you had to categorize it by application area, what category would you put it in? Why?

2. Critically evaluate Prograph in terms of the programming language criteria given/used in class. Give at least one (positive) criteria it satisfies, and at least one case where the language has a short-coming. Explain how Prograph meets or fails to meet the criteria you've chosen.

N. ***(14 marks)***

Consider a C++ class called *fraction* that implements a fraction as composed of separate denominator and numerator, both of type integer. Give declaration of the *fraction* class (in C++). Include definition of operators + and – , and the relation < . Define methods *numerator()* and *denominator()* (which return the numerator and denominator of the fraction, respectively). Don't forget constructors and destructors — if you need them.

Note: you do not have reduce the numerator and denominator after the + and – operations; i.e. if the fraction is not in simplest form, that is all right.